

Data Structures for Indexing Triple Table^{*}

Roman Meca, Michal Krátký, Peter Chovanec, and Filip Křížka

Department of Computer Science, VŠB – Technical University of Ostrava
Czech Republic

{roman.meca.st, michal.kratky, peter.chovanec, filip.krizka}@vsb.cz

Abstract. Semantic-based approaches are relatively new technologies. Some of these technologies are supported by specifications of W3 Consortium, i.e. RDF, SPARQL and so on. There are many areas where semantic data can be utilized, e.g. social networks, annotation of protein sequences etc. From the physical database design point of view, several index data structures are utilized to handle this data. In many cases, the well-known B-tree is used as a basic index supporting some operations. Since the semantic data are multidimensional, a common way is to use a number of B-trees to index the data. In this article, we review other index data structures; we show that we can create only one index when we utilize a multidimensional data structure like the R-tree. We compare a performance of the B-tree indices with the R-tree and some its variants. Our experiments are performed over a huge semantic database, we show advantages and disadvantages of these data structures.

1 Introduction

Semantic-based approaches are new technologies trying to allow computers to handle semantic information. These approaches have many specific applications. The main advantage of a semantic system is that the computer can reveal unexpected facts, e.g. a new effective drug combination in medicine [6], unexpected relationships in social networks [19] or artificial intelligence [28] can be discovered. This is the reason why the semantic systems are a current research topic.

The W3 Consortium have released some specifications related to semantic technologies¹, e.g. RDF [32] as a model of the semantic data or SPARQL [25] as a query language for the RDF data. In addition, more general specifications are also usable for a semantic DBMS supporting SPARQL or another query language, e.g. XML [9] or WSDL [12] for a communication with the DBMS.

In this article, we also list a lot of semantic DBMS with the query languages they support and the indices they utilize. Since the semantic DBMS often utilize a relational DBMS as a storage for the RDF triple table (representing the RDF data), the B-tree [13] is often used as the main index. The main issue of this

^{*} This work is partially supported by SGS, VŠB – Technical University of Ostrava, No. SP2015/192 and No. SP2015/170, Czech Republic.

^{**} This work is an extended version of an article accepted at the DATESO workshop.

¹ http://www.w3.org/standards/techs/rdf#w3c_all

physical implementation is that a number of B-trees have to be built to support queries over the triple table. However, there are other index data structures capable to handle semantic data. In this article, we show that it is possible to create only one index if we utilize a multidimensional data structure like the R-tree [22] or some of its variants (namely the Signature R-tree [30] or the Ordered R-tree [31]).

In Section 2, we present some basic terms related to semantic technologies. In Section 3, we describe the basic physical design for the RDF data. Section 4 describes some negative issues of the B-tree as an index data structure for the triple table. In addition, the R-tree, R*-tree [8], and two their variants are described. In Section 5, we summarize the advantages and disadvantages of these data structures for various queries over the LUBM data collection [21]. Finally, we conclude the article and outline the possibilities of our future work.

2 Semantic Technologies

In this section, we briefly introduce a theoretical basis of the RDF model [32] and the SPARQL query language [25] standardized by W3C. We recommend the book [20] for a more detailed review.

2.1 RDF Model

RDF (Resource Description Framework) is a general model representing information on the Web; data are modeled as a directed labeled graph [32]. Each edge represents a relationship between an *object* and a *subject*: two nodes of the graph. The label of the edge is called *property*. An example of the graph is given in Figure 1. This tuple (subject, property, object) is called an *RDF triple* (s,p,o).

The values of each triple usually include IRI (Internationalized Resource Identifiers) [15] identifying an abstract or a physical resource. In [20], the author introduces the following definition:

Definition 1 (RDF triple). *Let us assume there are pairwise disjoint infinite sets I , B , and L , where I represents the set of IRIs, B the set of blank nodes, and L the set of literals. We call a triple $(s, p, o) \in (I \cup B)I(I \cup B \cup L)$ an *RDF triple*, where s represents the subject, p the predicate, and o the object of the *RDF triple*.*

A *triple table* is a set of RDF triples; it is a representation of the RDF graph. In Table 1, we see a fragment of the triple table to the RDF graph in Figure 1. A triple store or an RDF database is an engine enabling to store an RDF graph and efficient processing of queries. However, we usually require other operations like update, insert or delete.

Some RDF stores add a fourth element to the triple; this fourth element contains the context of the triple [14]. There are RDF engines enabling to manage these quads [16].

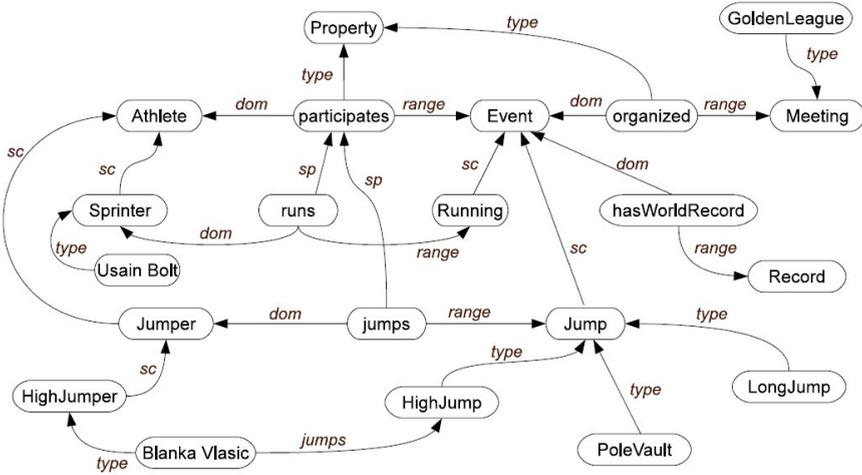


Fig. 1. An example of an RDF graph [18]

Subject	Property	Object
LongJump	type	Jump
Blanka Vlastic	jumps	HighJump
GoldenLeague	type	Meeting

Table 1. A fragment of an RDF triple table [18]

The RDF specification [32] does not define any way how to store and index the triple table; therefore, there are many variants of the physical design of the triple table and we describe them in Section 3.

2.2 SPARQL Query Language

Although there are many query languages for RDF data², e.g. SPARQL/Update (or SPALUR) [38], SPARQL 1.1 [25] is a de-facto standard query language for RDF data. It is similar to SQL in many features. SPARQL 1.1 also includes insert, update, and delete operations.

The basic query construct of the SELECT statement includes SELECT <projection> WHERE <sequence of triple patterns>. A variable in SPARQL defined by the symbol ? and a name represents the main difference compared to SQL; they define unknown values of o , s or p in a pattern as well as a relationship among triple patterns. We distinguish four types of the SPARQL query (for more details see [25]):

- *SELECT* – returns the result relation defined by the projection and patterns.

² <http://www.w3.org/2001/11/13-RDF-Query-Rules/>

- *ASK* – similar to the *SELECT* query; however, it returns the boolean value; *true* if the result is not empty, otherwise *false*.
- *CONSTRUCT* – allows to format own result graph over the triples returned by the patterns.
- *DESCRIBE* – returns the node (and its neighbours) defined by the patterns.

A form of `<pattern>` determines the selectivity of a query over the triple table. We can distinguish a point query ((s, p, o) returning 0 or 1 triple, or a range query where the query ($(s, *, *)$) can return more triples than the query ($(s, p, *)$).

Example 1 (SPARQL Queries).

1. `SELECT ?s ?p ?o WHERE { ?s ?p ?o }`
This query selects the whole triple table, it represents the range query $(*, *, *)$.
2. `SELECT * WHERE { <Blanka Vlastic> <jumps> <HighJump> }`
`ASK { <Blanka Vlastic> <jumps> <HighJump> }`
These two queries are similar; the *SELECT* query returns 0 or 1 triple, on the other hand, the *ASK* query returns true in the case the triple exists in the graph. These queries represent the point (s, p, o) query over the triple table.
3. `SELECT ?s WHERE { ?s <type> <Jump> }`
`ASK { ?s <type> <Jump> }`
`CONSTRUCT ?s <type> <Discipline> WHERE { ?s <type> <Jump> }`
These three queries include the same selection: the range query $(*, <type>, <Jump>)$. The *SELECT* query returns all subjects matched by the range query, the *ASK* query returns true if any triple exists in the graph, and the *CONSTRUCT* query returns triples $(*, <type>, <Discipline>)$ for all triples retrieved by the selection.
4. `SELECT ?p ?o WHERE { <organized> ?p ?o }`
This query selects all triples matched by the range query $(<organized>, *, *)$. The selectivity of this query is probably lower than the selectivity of the queries 2 and 3; however, it is higher compared to the query 1.

Moreover, the selection includes zero or more join operations. In Figure 2, we show two queries including more join operations. A query with one join is shown in Figure 2(a). In this *SELECT*, we can see two output variables $o1$ and $o2$. In Lines 2 and 3, the range queries $(*, <type>, *)$ and $(*, <jumps>, *)$ are defined. Results of these range queries are then joined using the subject represented by the j variable and objects for variables $o1$ and $o2$ are returned.

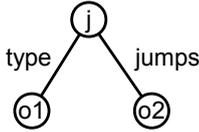
A more complex SPARQL query with join is shown in Figure 2(b). This *SELECT* also contains the output variables $o1$ and $o2$. However, this query is evaluated by a sequence of three joins: the first join involves sets defined by queries in Lines 2 and 3, the second join involves the result of the previous join and the result of the query in Line 4, and the last join involves the result of the previous join and the result of the query in Line 5. The result of the complete query includes subjects and objects for the variables s and o .

```

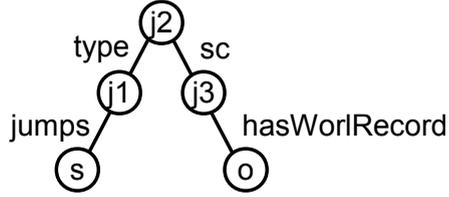
1. SELECT ?o1 ?o2 WHERE {
2.   ?j <type> ?o1 .
3.   ?j <jumps> ?o2
4. }

1. SELECT ?s ?o WHERE {
2.   ?s <jumps> ?j1 .
3.   ?j1 <type> ?j2 .
4.   ?j2 <sc> ?j3 .
5.   ?j3 <hasWorldRecord> ?o
6. }

```



(a)



(b)

Fig. 2. Two SPARQL query with join and their graph representations

3 Existing Triple Stores

In Table 2, we show triple stores introduced from 2002 to 2014. These triple stores include academic prototypes, commercial solutions as well as open source projects. Although some details of their implementation are not known, we can distinguish three basic types of the physical design for the triple table [18]:

1. *Triple Table (TT)* – in this case, triples are stored in a sequence array.
2. *Property Table (PT)* – in this case, we define a tuple $(s, o_1, o_2, \dots, o_n)$ for properties p_1, p_2, \dots, p_n . Tuples of this schema are stored in a sequence array. We can define more property tables in that cases the number of properties is higher than n .
3. *Vertical Partitioning (VP)* – the property table where $n = 1$.

Except these main approaches there are also some other variants and improvements, for example Hierarchical Property Partitioning utilized in roStore [17]. In some works, we distinguish the *Multiple indices* approach, which means that some combinations of various indices together with a modification of the above described types are depicted. In Table 2, we can see the B-tree and its variants are the most commonly used data structure indexing the triple table.

³ <http://www.guha.com/rdfdb/>

⁴ <http://rdfstore.sourceforge.net/>

⁵ <http://www.bigdata.com/>

Store	Published	Last update	Physical design type	Index data structure	Supported query language
JENA [34]	2002	2014	PT	Hash-table, B-tree	SPARQL
RDFSuite [3]	2001	2003	PT	B-tree	SPARUL
Sesame [10]	2002	2014	PT	B-tree	SPALUR
3store [23]	2003	2013	TT	Hash-table	RDQL/SPARQL
rdfDB ³	2004	2010	TT	B-tree	SPARQL
RDFStore ⁴	2004	2006	TT	BerkeleyDB	SPARQL
Redland [7]	2002	2014	TT	Hash-table	SPARQL
AllegroGraph[1]	2006	2014		B-tree	SPARQL
sw-Store[2]	2009	2014	VP	B-tree, Bitmap	SPARQL
4store [24]	2009	2013	PT	Hash-table	SPALUR
YARS [26]	2005	2006	MI	B-tree	N3 extension
YARS2 [27]	2007		MI	Sparse index, B-tree	SPARQL
Kowari [42]	2005	2005	MI	AVL tree, B-tree	iTQL/RDQL
Hexastore [41]	2008		MI	B-tree	SPARQL
RDFJoin [35]	2008		VP	B-tree	SPARQL
RDFKB [36]	2009		MI	B-tree	-
BitMat [4]	2009	2013	MI	3D Bitmap	SPARQL-like
RDF-3X [37]	2008	2013	MI	B-tree	SPARQL
Parliament [29]	2009	2014	MI	B-tree, Heap table	-
Virtuoso[16]	2009	2014	MI	B-tree, Bitmap	SPALUR
RDFCube[33]	2007		MI	3D Hash-table	-
GRIN [40]	2007		MI	B-tree	SPARQL
BigData ⁵	2008	2014	MI	B-tree	SPARQL 1.1
Oracle [11]	2005	2014	MI	B-tree, R-tree	SPARQL
Marmotta [5]	2013	2014	MI	B-tree	SPARQL

Table 2. Triple Stores. TT - triple table PT - property table VP - vertical partitioning MI - multiple indices

4 Index Data Structures

4.1 B-tree

The B-tree is an one-dimensional paged data structure supporting point and one-dimensional range queries as well as update operations [13]. As result, in the case we want to support a general range query without a sequential scan of all leaf nodes, we have to create more indices.

For example, in the case of a B-tree with the compound key (s, p, o) , we can effectively utilize range queries $(s, p, *)$ and $(s, *, *)$. On the other hand, fast processing of the range query $(*, p, o)$ demands a sequential scan over all leaf nodes of the B-tree. To cover all combination of searched dimensions with efficient range query execution, three B-trees have to be created (see Table 3). Consequently, this solution means that the size of indices is probably higher than the table size. This issue is even more evident in the case of the Quad table; in Table 4, we see that we need 6 indices to cover all range queries over quads. There are two problematic issues related to this technique: the higher space overhead and the additional overhead of the update operations since more indices have to be updated.

		Compound key of the B-tree		
		(s, p, o)	(o, s, p)	(p, o, s)
Supporting range queries	(s, p, o)	(s, p, o)	(o, s, p)	(p, o, s)
	$(s, p, *)$	$(s, p, *)$	$(o, s, *)$	$(p, o, *)$
	$(s, *, *)$	$(s, *, *)$	$(o, *, *)$	$(p, *, *)$
	$(*, *, *)$	$(*, *, *)$	$(*, *, *)$	$(*, *, *)$

Table 3. B-tree indices for the triple table

		Compound key of the B-tree					
		(s, p, o, c)	(p, o, c)	(o, c, s)	(c, s, p)	(c, p)	(o, s)
Supporting range queries	(s, p, o, c)	(s, p, o, c)	$(p, *, *)$	$(o, *, *)$	$(c, *, *)$	(c, p)	(o, s)
	$(s, p, o, *)$	$(s, p, o, *)$	$(p, o, *)$	$(o, c, *)$	$(c, s, *)$		
	$(s, p, *, *)$	$(s, p, *, *)$	(p, o, c)	(o, c, s)	(c, s, p)		
	$(s, *, *, *)$						
	$(*, *, *, *)$						

Table 4. B-tree indices for the quad table

4.2 R-tree

Since the multidimensional R-tree [22] supports a general multidimensional range query, we can use it as a solution of the above mentioned problems instead of

a sequence scan in the B-tree. The R-tree can be thought of as an extension of the B-tree in a multidimensional space. It corresponds to a hierarchy of nested n -dimensional *minimum bounding rectangles* (MBR). If \mathcal{N} is an interior node, it contains couples of the form (R_i, P_i) , where P_i is a pointer to a child of the node \mathcal{N} . If R is its MBR, then the rectangles R_i corresponding to the children \mathcal{N}_i of \mathcal{N} are contained in R . Rectangles at the same tree level can overlap. If \mathcal{N} is a leaf node, it contains couples of the form (R_i, O_i) , so called *index records*, where R_i contains a spatial object O_i .

The split algorithm has the significant affect on the index performance. Three split techniques (*Linear*, *Quadratic*, and *Exponential*) proposed in [22] are based on a heuristic optimization. The Quadratic algorithm has turned out to be the most effective and other improved versions of R-trees are based on this method. An MBR can overlap another MBR in the same level of the tree; the probability increases linearly with increasing data dimension. This effect is known as *curse of dimensionality* [43].

There are many variants of the R-tree, e.g. R*-trees [8], R⁺-tree [39]. The R*-tree [8] differs from the R-trees mainly in the insertion algorithm. Although original R-tree algorithms tried only to minimize the area covered by MBRs, the R*-tree algorithms try to minimize overlapping between MBRs at the same levels and maximize the storage utilization. The R⁺-tree [39] is a variant of the R-tree which allows no overlap between regions corresponding to nodes at the same tree level; however, an item can be stored in more than one leaf node.

Since some intervals of a range query include only one value in the case of the triple table, we call the query as the narrow range query [30]. Therefore, we utilize the Signature R-tree [30] allowing to handle the range query more efficiently than the R-tree and its variants. Moreover, we use the Ordered R-tree [31] since we can define an ordering of attributes. These data structures are described in the following sections.

4.3 Signature R-tree

The Signature R-tree [30] contains MBRs in inner nodes (we suppose point data in leaf nodes) and one signature related to each MBR. The signature is created for tuples inserted in the subtree related to each MBR. As result, we can use two types of filtering when a range query scans the tree: the first filtering method tests whether an MBR is intersected by a query rectangle and the second filtering method tests whether a signature can include tuples of the query. As result, the Signature R-tree reads a lower number of nodes during the range query processing. This R-tree variant is however proposed only for point data and narrow range queries.

4.4 Ordered R-tree

The Ordered R-tree [31] is a simple combination of the R-tree and the B-tree. It means, we can use a general multidimensional range query, however we can

define an ordering for tuples inserted in the tree. Evidently, we can define only one ordering in one tree. There are two consequences:

1. For some range queries (corresponding to ordering defined for the tree), all leaf nodes intersected by the query rectangle include only result tuples. It is not generally true for the R-tree and its variants, but the range query of the B-tree provides the same behaviour.
2. We get tuples of the result sorted and it is not necessary to sort them after the range query is processed.

In this article, we utilize mainly the first property.

5 Experiments

In our experiments⁶, we compare the B-tree, as the main index data structure utilized in semantic DBMS, with the R-tree⁷, Signature R-tree, and Ordered R-tree. All index data structures are implemented in C++⁸. We utilize a generated synthetic data collection called LUBM including 133,573,856 triples [21], the size of the text file is 22.2 GB.

Query Group	Type	Result set size	#Queries	#Iterations
1	Range query	$\langle 1; 1 \rangle$	6	10,000
2	Range query	$\langle 2; 1,000 \rangle$	6	50
3	Range query	$\langle 1,001; 1,000,000 \rangle$	6	1
4	Range query	$\langle 1,000,001; \infty \rangle$	6	1
5	Point query	$\langle 1; 1 \rangle$	33,234,949	1

Table 5. Specification of query groups

We test the performance of point and range queries processed over the index data structures when a SPARQL query is evaluated. We use 5 groups of queries determined by the selectivity (see Table 5)⁹. QG5 represents a sequence of point queries processed during a join operation. In the case of QG1 and QG2, it is necessary to repeat a sequence of queries since the processing time of one query is unmeasurable. The number of iterations is written in the column #Iteration of the table. The column #Queries contains a number of various queries in one query group.

⁶ We run our experiments on 2 x Intel Xeon E5 2690 2.9GHz and 300GB RAM memory, OS Windows Server 2008.

⁷ More precisely, the R*-tree has been tested.

⁸ A part of the RadegastDB framework developed by DBRG – <http://db.cs.vsb.cz/>

⁹ A complete list of queries can be found in http://db.cs.vsb.cz/TechnicalReports/indices_for_rdf_data-query.pdf

We built the B-trees, the R-tree, the Signature R-tree, and the Ordered R-trees for the test data collection¹⁰. In Table 6 and Figure 3, we see basic characteristics of these indices. Since these data structures include string ids instead of strings, a term index is built. In the case of the Ordered R-tree, we do not need more trees like in the case of the B-tree, however, in this article, we want to test whether it is possible to find an optimal ordering for the Ordered R-tree, therefore we build the tree for more orderings of the attributes. We can see that the B-tree size is up-to 3× higher than the size of the R-tree-based indices. The R-tree is build in 58% of the B-tree build time. On the other hand, the build time for other R-tree-based indices is up-to 2× less efficient compared to the B-tree.

Index Data Structure	#Nodes	Size [GB]	Build Time [s]	
Term index	4,543,671	8.67	3,794.7	
B-tree	(<i>s, p, o</i>)	4,465,853	8.51	
	(<i>p, o, s</i>)			
	(<i>o, s, p</i>)			
R-tree	1,495,289	2.85	2,228.1	
Signature R-tree	1,641,905	3.13	6,143.5	
Ordered R-tree	(<i>s, p, o</i>)	1,541,677	2.94	6,404.1
	(<i>p, o, s</i>)	1,499,602	2.86	7,193.5
	(<i>o, s, p</i>)	1,433,703	2.73	6,791.1
	(<i>s, o, p</i>)	1,541,677	2.94	7,232.2
	(<i>p, s, o</i>)	1,579,935	3.01	7,535.6
	(<i>o, p, s</i>)	1,429,151	2.73	6,933.0

Table 6. Statistics of index data structures

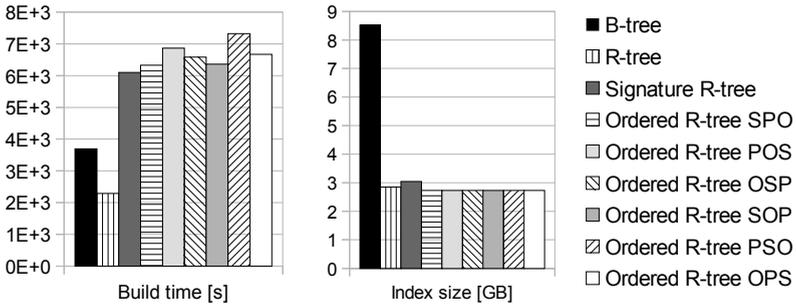


Fig. 3. Index build time and index size

¹⁰ The page size is 2,048 B for all data structures.

In Figure 4, we can see the query processing time for all query groups; the processing time is the average time of all queries in one group. Similarly, Figure 5 includes DAC for all query groups. Evidently, the B-tree provides the most efficient performance especially in the case of the higher selectivity. The reason of this result is the minimal DAC of the B-tree since only leaf nodes including result tuples are scanned. In the case of the lower selectivity (see GP4 in Figure 4), results of all index data structures are similar.

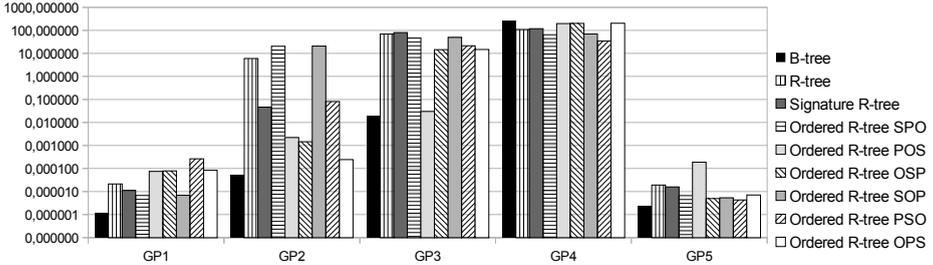


Fig. 4. Processing time for all query groups [s]

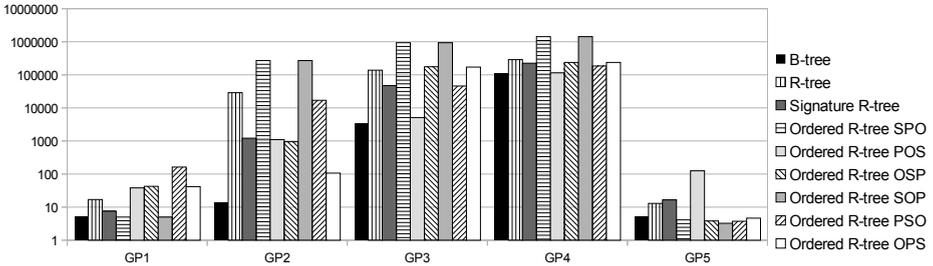


Fig. 5. DAC for all query groups

We see that the Signature R-tree and the Ordered R-tree outperform the R-tree in most cases. Although the average processing time of the Signature R-tree is lower compared to the Ordered R-tree, we can find a query in each query group where it exists an ordering of the Ordered R-tree such that the Ordered R-tree outperforms the Signature R-tree. Let us consider query processing times in Figure 6. In the case of Q1 ($S='AssociateProfessor'$, $P='type'$, $O=*$), the Ordered R-trees SPO and SOP outperform the Signature R-tree and other Ordered R-trees, however in the case of Q7 ($S=*$, $P='PublicationAuthor'$, $O='AssistantProfessor'$) the performance of these Ordered R-trees is the lowest. Similarly, in the case of Q11 ($S=*$, $P=*$, $O='Course2'$), the Ordered R-tree OPS outperforms other R-tree variants and its performance is the same as the

performance of the B-tree. Similarly, in the case of Q14 ($S=*$, $P='worksFor'$, $O=*$), the Ordered R-tree SOP outperforms other R-tree variants. However, we must keep in mind that this effect depends on a query and a concrete ordering of the Ordered R-tree.

Although, it is clear that the B-tree provides the most efficient processing time, there are some improvements of multidimensional data structures. The first one, the index size of a multidimensional data structure is up to $3\times$ lower the B-tree index size. The second one, in the case of the B-tree it is necessary to change ordering of values in a triple when a query processor want to use an index with different ordering than another index returns, it means an additional time overhead in this case.

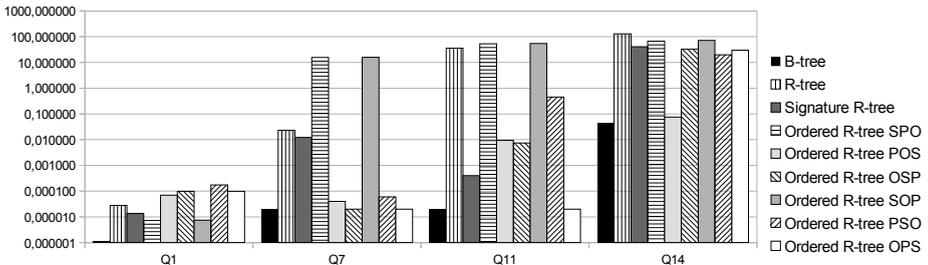


Fig. 6. Processing time for some queries [s]

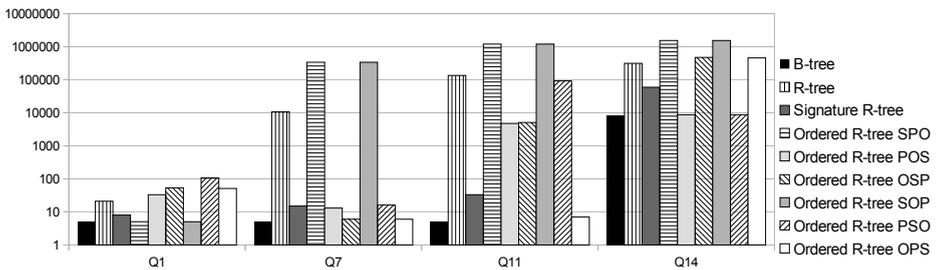


Fig. 7. DAC for some queries

As result, let us consider a workload including queries accessing the most tree nodes. If the cache size is lower than the number of B-tree nodes, a multidimensional data structure would provide the higher performance than the B-tree in the case the cache includes all nodes of the multidimensional data structure.

6 Conclusion

In this article, we compared the performance of the B-tree with the R-tree, the Signature R-tree, and the Ordered R-tree for the triple table and point and range queries processed during the evaluation of a SPARQL query. The Signature R-tree and the Ordered R-tree outperform the R-tree for most queries. Although the average processing time of the Signature R-tree is lower compared to the Ordered R-tree, in each query group, we can find a query where there is such an ordering of the Ordered R-tree outperforming the Signature R-tree.

The B-tree provides the most efficient processing time; the average processing time of the B-tree is 74% of the Signature R-tree's processing time. However, there are some specific improvements of multidimensional data structures. The first one, index size of a multidimensional data structures is up to $3\times$ lower than the B-tree index size. The second one, in the case of the B-tree it is necessary to change ordering of values in each triple when a query processor want to use an index with different ordering than another index returns. Consequently, it means an additional time overhead of the query processing.

References

- [1] J. Aasman. *Allegro Graph: RDF Triple Database*. Tech. rep. Technical Report 1, Franz Incorporated, 2006. URL: <http://www.franz.com/agraph/allegrograph/>.
- [2] D.J. Abadi et al. "SW-Store: a vertically partitioned DBMS for semantic web data management". In: *The VLDB Journal* 18.2 (2009), pp. 385–406.
- [3] S. Alexaki et al. "The ICS-FORTH RDFSuite: Managing voluminous RDF description bases". In: *Proceedings of 2nd International Workshop on the Semantic Web (SemWeb'01)*. 2001.
- [4] M. Atre, J. Srinivasan, and J.A. Hendler. *BitMat: A Main Memory RDF Triple Store*. Tech. rep. 2009. URL: http://www.cs.rpi.edu/~atrem/bitmat_techrep.pdf.
- [5] Reto Bachmann-Gmur. *Instant Apache Stanbol*. Packt Publishing Ltd, 2013. ISBN: 978-1-78328-123-7.
- [6] Amos Bairoch et al. "The universal protein resource (UniProt)". In: *Nucleic acids research* 33 (2005), pp. D154–D159.
- [7] D. Beckett. "The design and implementation of the Redland RDF application framework". In: *Computer Networks* 39.5 (2002), pp. 577–588.
- [8] Norbert Beckmann et al. "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles". In: *Proceedings of the ACM International Conference on Management of Data (SIGMOD 1990)*. Vol. 19. AMC, 1990, pp. 322–331.
- [9] Tim Bray et al. "Extensible markup language (XML)". In: *World Wide Web Journal* 2.4 (1997), pp. 27–66.
- [10] J. Broekstra, A. Kampman, and F. Van Harmelen. "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema". In: *Proceedings of the Semantic Web-ISWC*. Vol. 2342. Springer, 2002.

- [11] E.I. Chong et al. “An efficient SQL-based RDF querying scheme”. In: *Proceedings of 31th International Conference on Very Large Data Bases (VLDB 2005)*. VLDB Endowment. 2005, pp. 1216–1227.
- [12] Erik Christensen et al. *Web services description language (WSDL) 1.1*. Recommendation. W3C, 2001. URL: <http://www.w3.org/TR/wsdl>.
- [13] Douglas Comer. “Ubiquitous B-tree”. In: *ACM Computing Surveys (CSUR)* 11.2 (1979), pp. 121–137.
- [14] R. Cyganiak, A. Harth, and A. Hogan. *N-quads: Extending n-triples with context*. Tech. rep. 2008. URL: <http://sw.deri.org/2008/07/n-quads/>.
- [15] Martin Dürst and Michel Suignard. *Internationalized resource identifiers (IRIs)*. Tech. rep. RFC 3987, January, 2005. URL: <http://www.ietf.org/rfc/rfc3987.txt>.
- [16] Orri Erling and Ivan Mikhailov. “Virtuoso: RDF support in a native RDBMS”. In: (2010), pp. 501–519.
- [17] David Faye et al. “RDF triples management in roStore”. In: *Actes de IC2011* (2012), pp. 755–770.
- [18] David Célestin Faye, Olivier Curé, and Guillaume Blin. “A survey of RDF storage approaches”. In: *ARIMA Journal* 15 (2012). URL: <http://arima.inria.fr/015/015002.html>.
- [19] Tim Finin et al. “Social networking on the semantic web”. In: *Learning Organization journal* 12.5 (2005), pp. 418–435.
- [20] S. Groppe. *Data management and query processing in semantic web databases*. Springer, 2011. ISBN: 978-3-642-19356-9.
- [21] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. “LUBM: A benchmark for OWL knowledge base systems”. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 3.2 (2005), pp. 158–182.
- [22] Antonin Guttman. “R-trees: a dynamic index structure for spatial searching”. In: *Proceedings of the ACM International Conference on Management of Data, (SIGMOD '84)*. Vol. 14. 2. 1984, pp. 47–57.
- [23] S. Harris and D.N. Gibbins. “3store: Efficient bulk RDF storage”. In: *volume 89 of CEUR Workshop Proceedings* (2003).
- [24] S. Harris, N. Lamb, and N. Shadbolt. “4store: The design and implementation of a clustered RDF store”. In: *Proceedings of 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*. 2009, pp. 94–109.
- [25] S. Harris and A. Seaborne. “SPARQL 1.1 query language”. In: *W3C Recommendation* (2013). URL: <http://www.w3.org/TR/sparql11-query/>.
- [26] A. Harth and S. Decker. “Optimized index structures for querying rdf from the web”. In: *Proceedings of 3th Latin American Web Congress, (LA-WEB 2005)*. IEEE. 2005.
- [27] A. Harth et al. “Yars2: A federated repository for querying graph structured data from the web”. In: *The Semantic Web* 4825 (2007), pp. 211–224.
- [28] M Tim Jones. *Artificial Intelligence A System Approach*. Laxmi Publications, Ltd., 2008. ISBN: 978-0763773373.

- [29] D. Kolas, I. Emmons, and M. Dean. “Efficient linked-list rdf indexing in parliament”. In: *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*. Vol. 9. 2009, pp. 17–32.
- [30] Michal Krátký et al. “Efficient processing of narrow range queries in multi-dimensional data structures”. In: *Proceedings of 10th International Database Engineering and Applications Symposium, (IDEAS’06)*. IEEE. 2006.
- [31] Filip Křížka, Michal Krátký, and Radim Bača. “On support of ordering in multidimensional data structures”. In: *Proceedings of Advances in Databases and Information Systems (ADBIS 2010)*. Vol. 6295. LNCS. Springer. 2010, pp. 575–578.
- [32] Frank Manola, Eric Miller, Brian McBride, et al. “RDF primer”. In: *W3C recommendation 10* (2004). URL: <http://www.w3.org/TR/rdf-primer/>.
- [33] Akiyoshi Matono, SaidMirza Pahlevi, and Isao Kojima. “RDFCube: A P2P-Based Three-Dimensional Index for Structural Joins on Distributed Triple Stores”. In: *Databases, Information Systems, and Peer-to-Peer Computing*. Vol. 4125. LNCS. Springer, 2007. ISBN: 978-3-540-71660-0.
- [34] B. McBride. “Jena: A semantic web toolkit”. In: *Internet Computing, IEEE 6.6* (2002), pp. 55–59.
- [35] J.P. McGlothlin and L.R. Khan. *RDFJoin: A scalable data model for persistence and efficient querying of RDF datasets*. Tech. rep. 2009.
- [36] J.P. McGlothlin and L.R. Khan. “RDFKB: efficient support for RDF inference queries and knowledge management”. In: *Proceedings of the 2009 International Database Engineering & Applications Symposium*. ACM. 2009, pp. 259–266.
- [37] T. Neumann and G. Weikum. “RDF-3X: a RISC-style engine for RDF”. In: *Proceedings of the VLDB Endowment*. Vol. 1. 1. VLDB Endowment, 2008, pp. 647–659.
- [38] A. Seaborne et al. “SPARQL/Update: A language for updating RDF graphs”. In: *W3C Member Submission 15* (2008).
- [39] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. “The R⁺-Tree: A Dynamic Index for Multi-Dimensional Objects”. In: *Proceedings of 13th International Conference on Very Large Data Bases (VLDB 1997)*. Morgan Kaufmann, 1987.
- [40] Octavian Udrea, Andrea Pugliese, and VS Subrahmanian. “GRIN: A graph based RDF index”. In: *Proceedings of the 22nd national conference on Artificial intelligence, (AAAI’07)*. Vol. 1. 2007, pp. 1465–1470.
- [41] C. Weiss, P. Karras, and A. Bernstein. “Hexastore: sextuple indexing for semantic web data management”. In: *Proceedings of the VLDB Endowment 1.1* (2008), pp. 1008–1019.
- [42] D. Wood, P. Gearon, and T. Adams. “Kowari: A platform for semantic web storage and analysis”. In: *Proceedings of XTech 2005 Conference*. 2005.
- [43] Cui Yu. *High-Dimensional Indexing*. Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2002. ISBN: 3-540-44199-9.